

# 分布式系统互斥与幂等问题的探索与实践

刘云鹏@美团点评酒旅事业群 201703



# Agenda

---

Chapter1 : Distributed System Theory

Chapter2 : Mutex

Chapter2 : Idemponent



# Chapter1 : Distributed System Theory

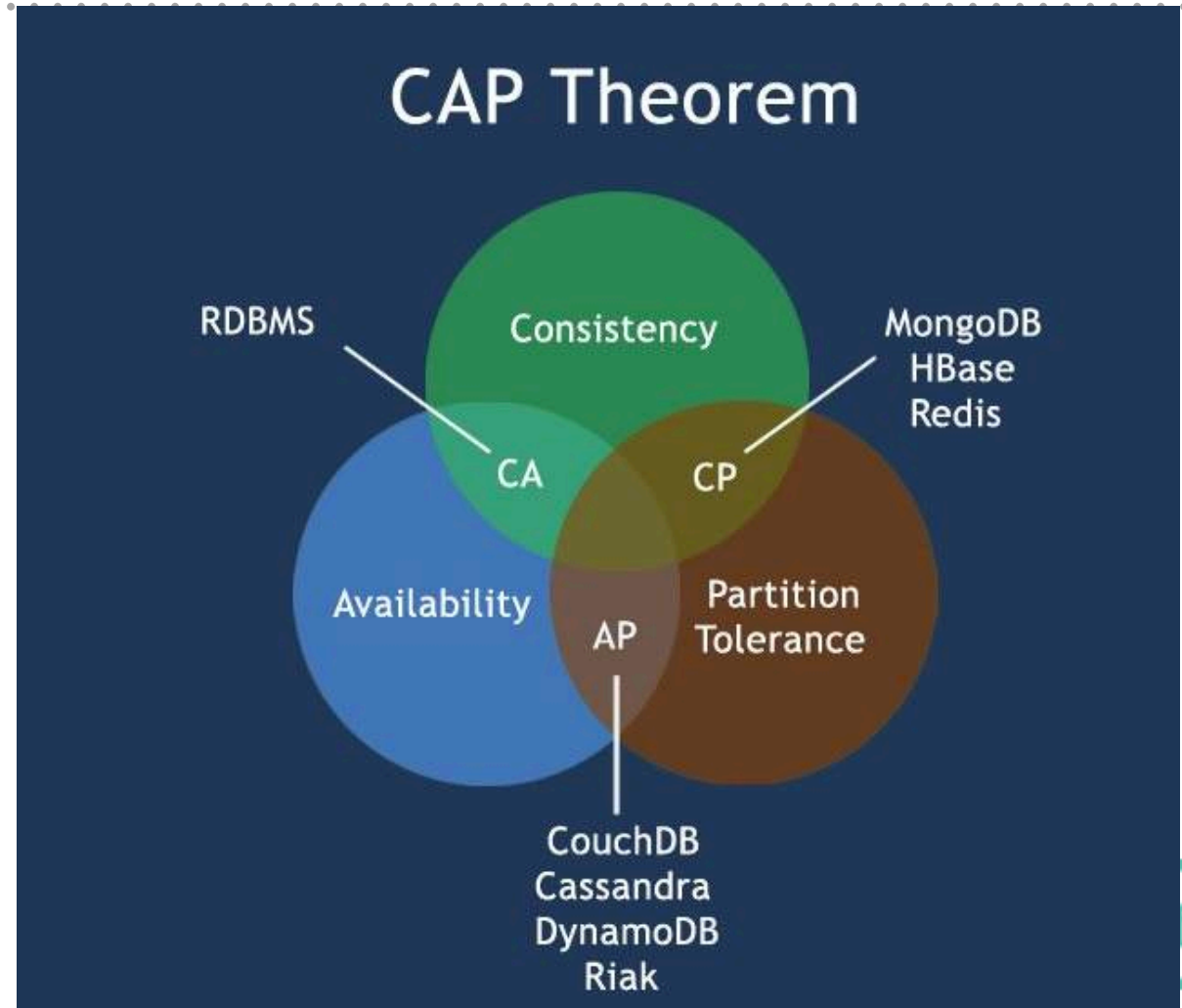


A distributed system  
is a collection of **independent** computers  
that behave **as a single coherent system**.

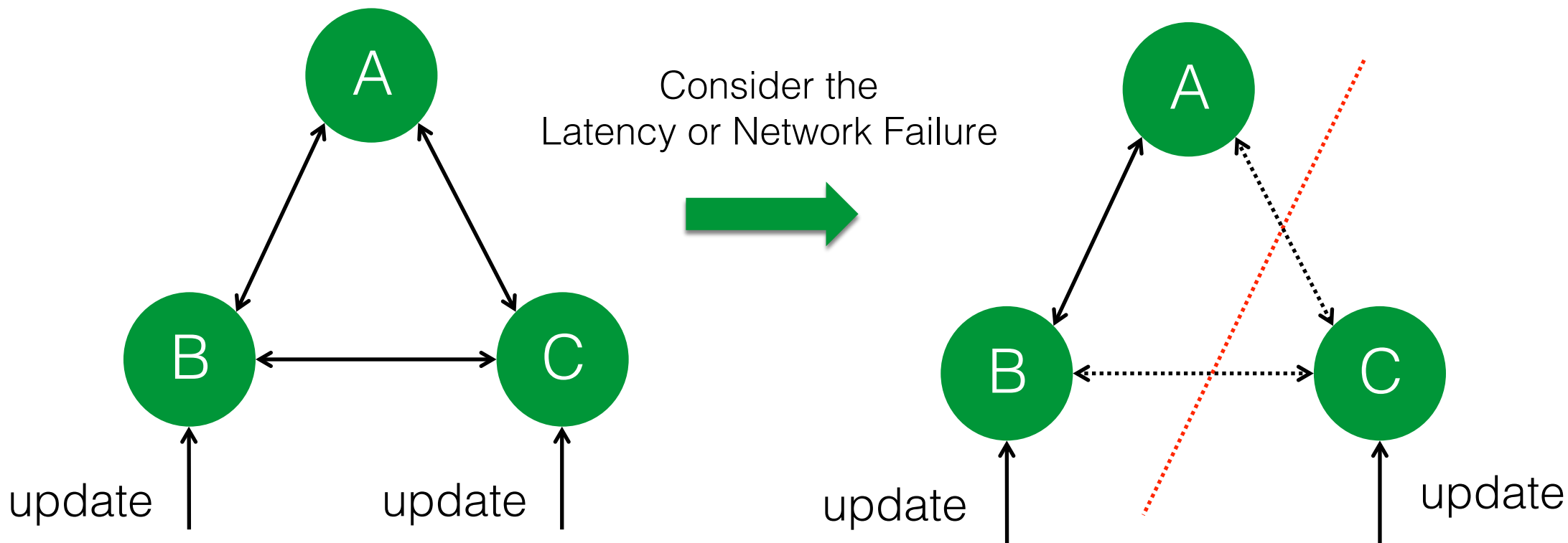


# CAP

- Eric Brewer , 1998
- Consistency
- Availability
- Partition Tolerance



# CAP



一旦出现分区：允许update，丧失C；反之，丧失A

In distributed system,  
partition tolerance is **not an option**,  
it's **required**.



Building reliable distributed systems at a  
worldwide scale demands **trade-offs** between  
**consistency** and **availability**

*-Werner Vogels*





# Consistency

---

- Client-side

Every read receives the most recent write or an error



# Consistency

---

- 强一致性 ( Strong Consistency )
  - 更新完成后，任何后续访问都将返回更新后的值
- 弱一致性 ( Weak Consistency )
  - 更新完成后，不保证返回更新后的值，通常会经历一个 “不一致窗口”
- 最终一致性 ( Eventual Consistency )
  - 弱一致性的特例，保证如果没有新的更新，在一段时间后（不一致窗口），将返回更新后的值
  - 因果一致性、读写一致性、会话一致性、单调读一致性、单调写一致性



# Strong Consistency vs Eventual Consistency



# Strong Consistency

---

- 数据库事务（ACID，基于关系型数据库）
- 分布式事务（JTA：两阶段提交）

必要条件：Mutex



# Eventual Consistency

---

- 本地信息表+消息日志
- 数据库+事务消息
- 消息补偿
- Conflict-free Replicated Data Type——CRDT

必要条件： Idempotent

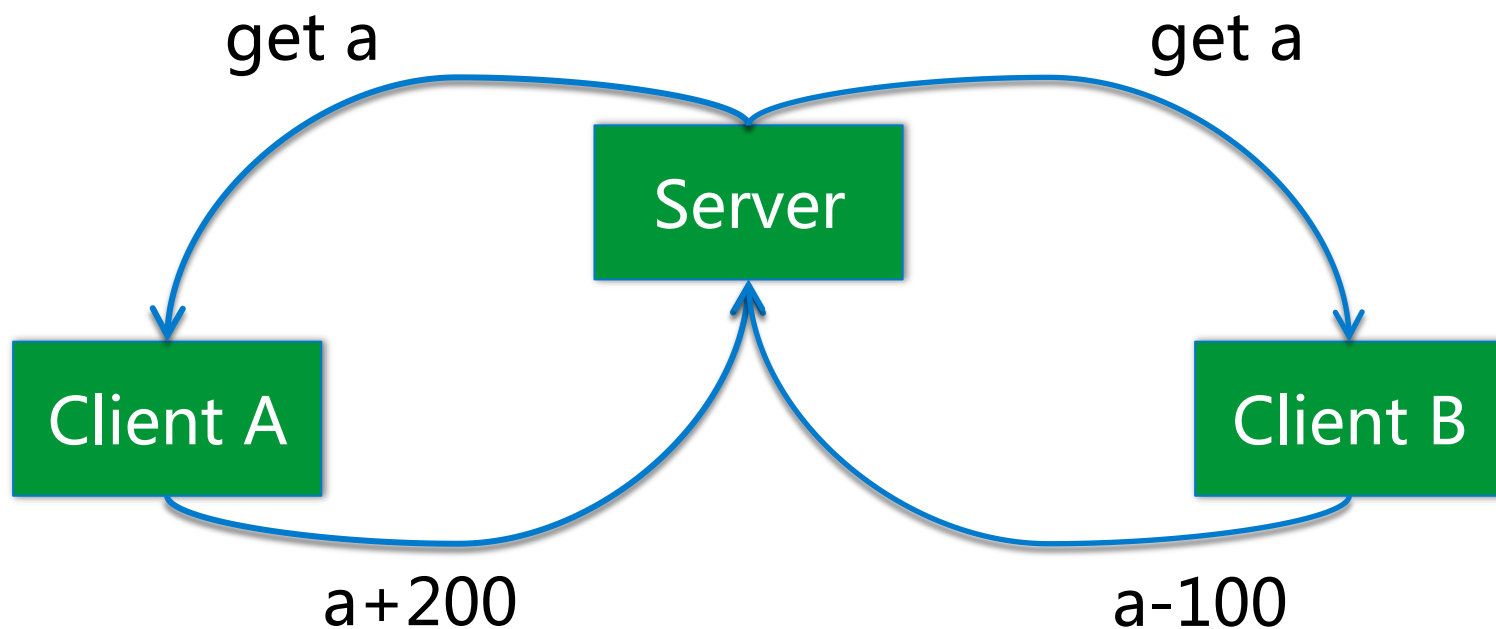


## Chapter2 : Mutex



# Why

---



先add , 再get  
最初  $a = 500$   
最终  $a = 600 ?$   
 $700 ?$   
 $400 ?$

**互斥是为了解决共享资源抢占问题**





# 互斥-实现方式

---

- 数据存储本身，如数据库事务
- 业务逻辑代码规避
- 分布式锁
- ...



分布式锁如何实现？



# 分布式锁-基本条件

---

## 条件一：锁存储空间

- 多线程：内存
- 多进程：共享内存或者磁盘
- 分布式环境：
  - 数据库（行锁、version乐观锁）
  - 外部缓存：Redis、Tair、Memcached...
  - Zookeeper
  - 集群中一台独立的主机



# 分布式锁-基本条件

---

## 条件二：锁需要被唯一标识

- 多线程：锁对象的引用
- 多进程：对象的引用或唯一文件名
- 分布式环境：
  - 全局唯一锁名称

# 分布式锁-基本条件

---

## 条件三：锁要有至少两种状态

- 多线程：*ReentrantLock*中的*status*，0表示没有线程竞争，大于0表示有线程竞争
- 多进程：信号量大于等于0表示可以进入临界区，小于0表示需要阻塞
- 分布式环境：
  - 0/1
  - 有锁/无锁
  - 存在/不存在



# 分布式锁-简易实现

---

数据库表，字段为锁的ID（唯一标识）和锁的状态（0表示没有被锁，1表示被锁）

伪代码：

```
lock = mysql.get(id);  
while(lock.status == 1) {  
    sleep(100);  
}  
mysql.update(lock.status = 1)  
doSomething();  
mysql.update(lock.status = 0);
```



问题？

# 分布式锁-问题

---

问题1：锁状态判断原子性无法保证

- 目标：保证锁状态判断的原子性

问题2：网络断开或主机宕机，锁状态无法清除

- 目标：在持有锁的主机宕机或者网络断开的时候，及时的释放掉这把锁

问题3：无法保证释放的是自己上锁的那把锁

- 目标：确定自己解的这个锁正是自己锁上的



# 分布式锁-进阶条件

---

- 可重入：线程中的可重入，指的是外层函数获得锁之后，内层也可以获得锁。
- 惊群效应（Thundering Herd）：在分布式锁中，惊群效应指的是，在有多个请求等待获取锁的时候，一旦占有锁的线程释放之后，所有等待的方都同时被唤醒，尝试抢占锁。
- 公平锁和非公平锁：不同的需求，可能需要不同的分布式锁。非公平锁普遍比公平锁开销小，但是有时候必须要实现公平锁。
- 阻塞锁和自旋锁：阻塞锁会有上下文切换，如果并发量比较高且临界区的操作耗时比较短，那么造成的性能开销就比较大。但是如果临界区操作耗时比较长，一直保持自旋，也会对CPU造成更大的负荷。





# 分布式锁-Zookeeper实现

---

- 顺序节点：ZooKeeper集群按照创建的顺序来创建节点，分别标记为001、002...
- 临时节点：客户端与ZooKeeper集群断开连接，则该节点自动被删除

## 分布式锁的基本逻辑：

1. 客户端调用create()方法创建名为“dlm-locks/lockname/lock-”的临时顺序节点。
2. 客户端调用getChildren(“lockname”)方法来获取所有已经创建的子节点。
3. 客户端获取到所有子节点path之后，如果发现自己的在步骤1中创建的节点是所有节点中序号最小的，那么就认为这个客户端获得了锁。
4. 如果创建的节点不是所有节点中需要最小的，那么则监视比自己创建节点的序列号小的最大的节点，进入等待。直到下次监视的子节点变更的时候，再进行子节点的获取，判断是否获取锁。



```

public final void lock() {
    if (checkReentrancy())
        return;

    .....
    String lockNode;
    try {
        lockNode = zk.create(getBaseLockPath(), ..., ..., CreateMode.EPHEMERAL_SEQUENTIAL);
        while (true) {
            .....
            localLock.lock();
            try {
                boolean acquiredLock = tryAcquireDistributed(zk, lockNode, true);
                if (!acquiredLock) {
                    condition.awaitUninterruptibly();
                } else {
                    locks.set(new LockHolder(lockNode));
                    return;
                }
            } finally {
                localLock.unlock();
            }
        }
    } catch .....
    }
}

```



```
private boolean checkReentrancy() {
    LockHolder local = locks.get();
    if (local != null) {
        local.incrementLock();
        return true;
    }
    return false;
}
```

```
protected boolean tryAcquireDistributed(ZooKeeper zk, String lockNode, boolean watch) throws ...{
    .....
    String myNodeName = lockNode.substring(lockNode.lastIndexOf('/') + 1);
    int myPos = lockList.indexOf(myNodeName);

    int nextNodePos = myPos - 1;
    while (nextNodePos >= 0) {
        Stat stat;
        if (watch)
            stat = zk.exists(baseNode + "/" + lockList.get(nextNodePos), signalWatcher);
        else
            stat = zk.exists(baseNode + "/" + lockList.get(nextNodePos), false);
        if (stat != null) {
            return false;
        } else {
            nextNodePos--;
        }
    }
    return true;
}
```



# 分布式锁-Zookeeper实现

---

开源Menagerie实现：<https://github.com/sfines/menagerie>

- 可重入锁：ThreadLocal存储次数
- 公平锁：JUC的condition
- 基于zookeeper的分布式锁是目前**最普遍、正确性最高**的实现方案



# 分布式锁-Redis实现

---

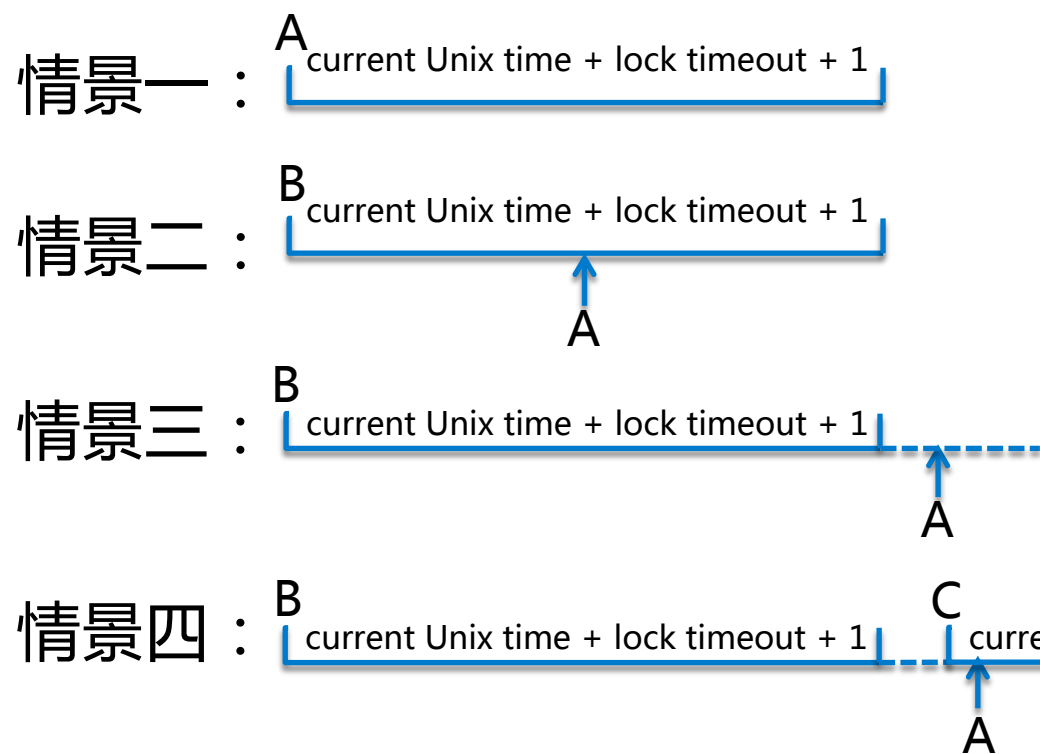
典型的实现是：SETNX+GETSET 或SETNX+EXPIRE

- SETNX：SET if Not eXists；原子性操作
- GETSET：先写新值，返回旧值；原子性操作
- Value中存储时间戳：防止宕机或网络断开造成死锁

基于单实例实现



# 分布式锁-Redis实现



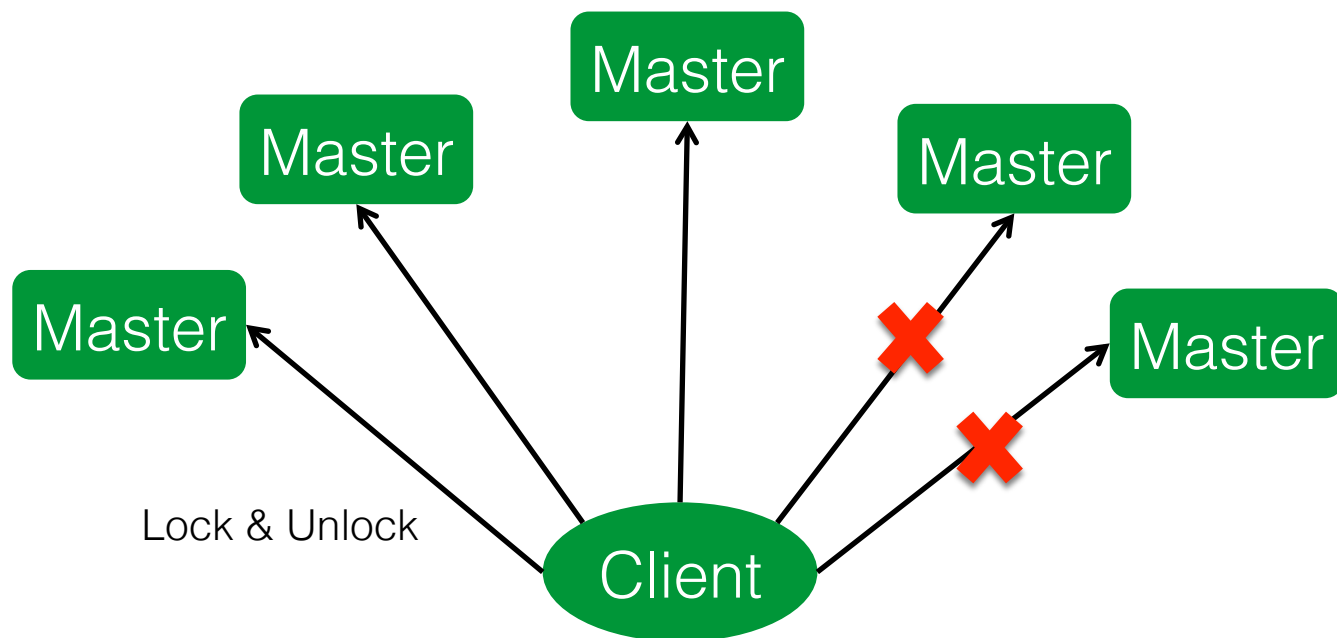
1. 线程A发送SETNX lock.orderid <current Unix time + lock timeout + 1> 尝试获得锁，如果锁不存在，则set并获得锁
2. 如果锁存在，则再判断锁的值（时间戳）是否大于当前时间，如果没有超时，则等待一下再重试。
3. 如果已经超时了，在用GETSET lock.{orderid} <current Unix time + lock timeout + 1>来尝试获取锁，如果这时候拿到的时间戳仍旧超时，则说明已经获得锁了。
4. 如果在此之前，另一个线程C快一步执行了上面的操作，那么A拿到的时间戳是个未超时的值，这时A没有如期获得锁，需要再次等待或重试。但是尽管A没拿到锁，但它改写了C设置的锁的超时值。但因为解锁并不是依据超时值，所以影响不大。

问题？



# 分布式锁-Redis实现

- Redisson : 官方客户端。 <https://github.com/mrniko/redisson>
- Redlock : Redis官方的多master节点分布式锁解决方案



多实例版本为Redlock

# 分布式锁-Tair实现

---

原理类似Redis

- expireLock：原子性。通过锁状态和过期时间戳来共同判断锁是否存在，只有锁已经存在且没有过期的状态才判定为有锁状态。在有锁状态下，不能加锁，能通过大于或等于过期时间的时间戳进行解锁。无需在value中存储时间戳，避免主机时钟不一致。





# Cerberus分布式锁

---

Cerberus：灵活可靠的多引擎分布式锁



# Cerberus分布式锁

---

## 特点一：多引擎

- Cerberus分布式锁使用了多种引擎实现方式（Tair、zookeeper、Redis），支持用户结合业务实际自主选择引擎

	Tair	Redis	ZK
并发量高	✓	✓	
响应时间敏感	✓	✓	
公平锁			✓
非公平锁	✓	✓	
读写锁			✓



# Cerberus分布式锁

---

## 特点二：接口统一

- 适配器模式屏蔽底层引擎实现细节。

a ) `void lock();`

获取锁，如果锁被占用，将禁用当前线程，并且在获得锁之前，该线程将一直处于阻塞状态。

b ) `boolean tryLock();`

如果锁可用，则获取锁，并立即返回值 `true`。如果锁不可用，则此方法将立即返回值 `false`。

c ) `boolean tryLock(long time, TimeUnit unit) throws InterruptedException;`

如果在给定时间内锁可用，则获取锁，并立即返回值 `true`。如果在给定时间内锁一直不可用，则此方法将立即返回值 `false`。

d ) `void lockInterruptibly() throws InterruptedException;`

获取锁，如果锁被占用，则一直等待直到线程被中断或者获取到锁

e ) `void unlock();`

释放当前持有的锁



# Cerberus分布式锁

---

## 特点三：支持降级

- Best effort，只提供接口，降级由业务实现

a) String switchEngine()

转换分布式锁引擎，按配置的引擎的顺序循环转换。

返回值：返回当前的engine名字，如："zk"。

b) String switchEngine(String engineName)

转换分布式锁引擎，切换为指定的引擎。

参数：engineName - 引擎的名字，同配置bean的名字，"zk"/"tair"。

返回值：返回当前的engine名字，如："zk"。

切换过程不转移状态！




# Cerberus分布式锁-错误示例

---


```
@Autowired
private IDistributedLockManager dlm;

public void lockInterruptibly() {
    Lock firstLock = dlm.getReentrantLock("test1");
    try {
        firstLock.lockInterruptibly();
        // synDoSomething();
    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        firstLock.unlock();
    }
}
```



```
@Autowired
private IDistributedLockManager dlm;

public void lockInterruptibly() throws InterruptedException {
    Lock firstLock = dlm.getReentrantLock("test1");
    firstLock.lockInterruptibly();
    try {
        // synDoSomething();
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        firstLock.unlock();
    }
}
```



# Cerberus分布式锁-现状

---

至今

- 分布式锁已持续迭代13个版本
- 先后稳定运行在了69个项目中：
  - 酒旅平台TDC
  - 酒旅平台结算
  - 酒旅跟团游
  - 酒店后台动态价格
  - 到店商户应用
  - .....



# Chapter3 : Idempotent



## Idempotent

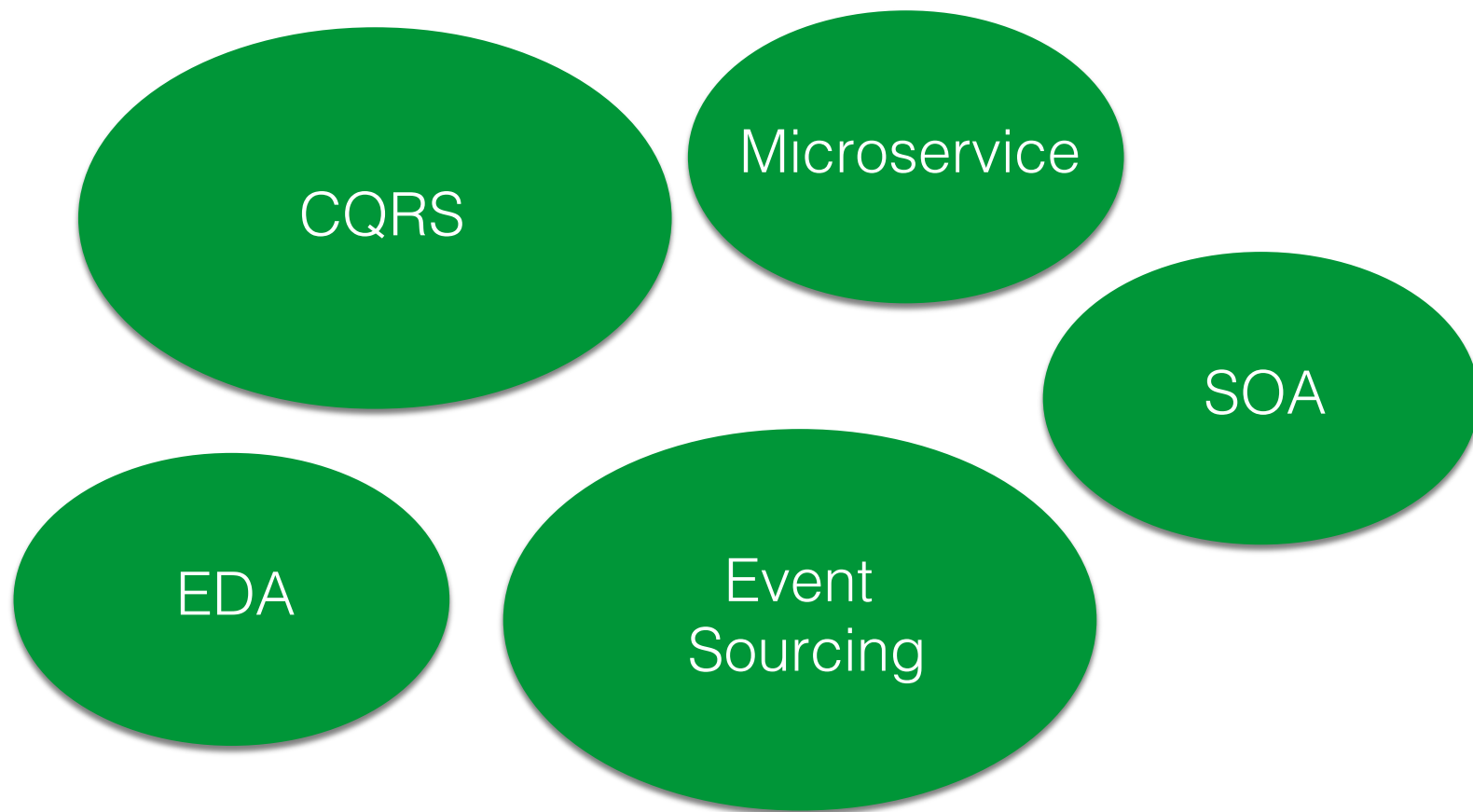
is used to describe an **operation** that will produce the **same results** if executed **once** or **multiple** times





# Why

---



- RPC
- Message

# Why

---

- 作为RD，你是否曾被以下问题困扰：
  - 某接口被莫名其妙调用了多次，导致许多重复工单生成。
  - 某结算流水信息重复生成。
  - MQ出问题了，消息丢失了。。
  - MQ恢复了，但是重复多发了好多数据。。
  - 我用了事件驱动，事件回放的时候许多之前成功执行的操作又特么执行了一遍。。





Within the context of a distributed system, you **cannot** have **exactly-once** message delivery.

*-Tyler Treat*



# Why

---

- 作为一名职业RD，我有如下建议：
  - 如果接口特别重要，建议你考虑一下**幂等**
  - 如果接口不是特别重要的话。。。你懂的



# Why

---

- 操作分两种：
  - 幂等：查询操作、无状态修改
  - 非幂等：除了上述都是。。

我们的目标是把非幂等操作“山寨”成幂等操作！



# GTIS

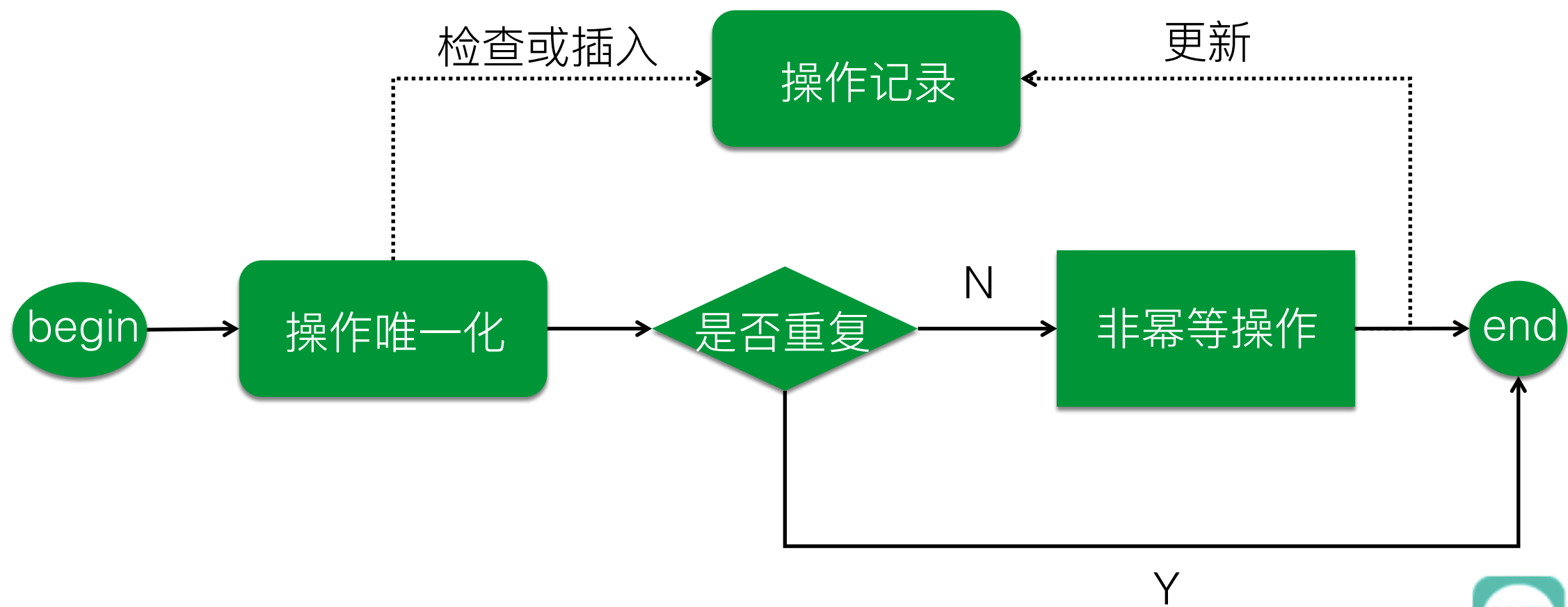
---

Gtis：轻量的重复操作关卡系统，间接实现操作幂等性。它具有如下特点：

- 高效：低延时，平均响应时间2ms内
- 简单：接入简捷方便，学习成本低
- 灵活：提供多种接口参数、使用策略，以满足不同需求
- 可靠：高可用降级；应用鉴权，业务隔离



# GTIS-基本原理





# GTIS-基本原理

---

空域唯一性

- 操作ID：MD5(应用名+服务+方法+业务自定义数据)。

一次操作=服务方法+传入的业务数据



# GTIS-基本原理

---

时域唯一性

- 操作ID的时效

你希望在规定时间内保证某次操作的幂等？



# GTIS-基本原理

---

- 数据结构：
  - 操作ID：唯一标识一次操作
  - 操作内容：业务数据、时间戳等用于检验的数据
- 存储引擎：
  - **Nosql**：tair、redis
  - 关系数据库：mysql



# GTIS-处理流程

---

- 1.业务方在业务操作之前，生成一个能够唯一标识该操作的transContents，传入GTIS；
- 2.GTIS根据传入的transContents，用MD5生成全局操作ID；
- 3.GTIS将全局ID作为key，current\_timestamp+transContents作为value放入Tair进行setNx，将结果返回给业务方；
- 4.业务方根据返回结果确定能否开始进行业务操作；若能，开始进行操作；若不能，则结束当前操作；
- 5.处理结束后，业务方将操作结果和请求结果传入GTIS，系统进行一次请求结果的检验；若成功，GTIS根据key取出value值，跟返回结果进行比对，如果相等，则将该全局ID的过期时间改为较长时间；
- 6.结束



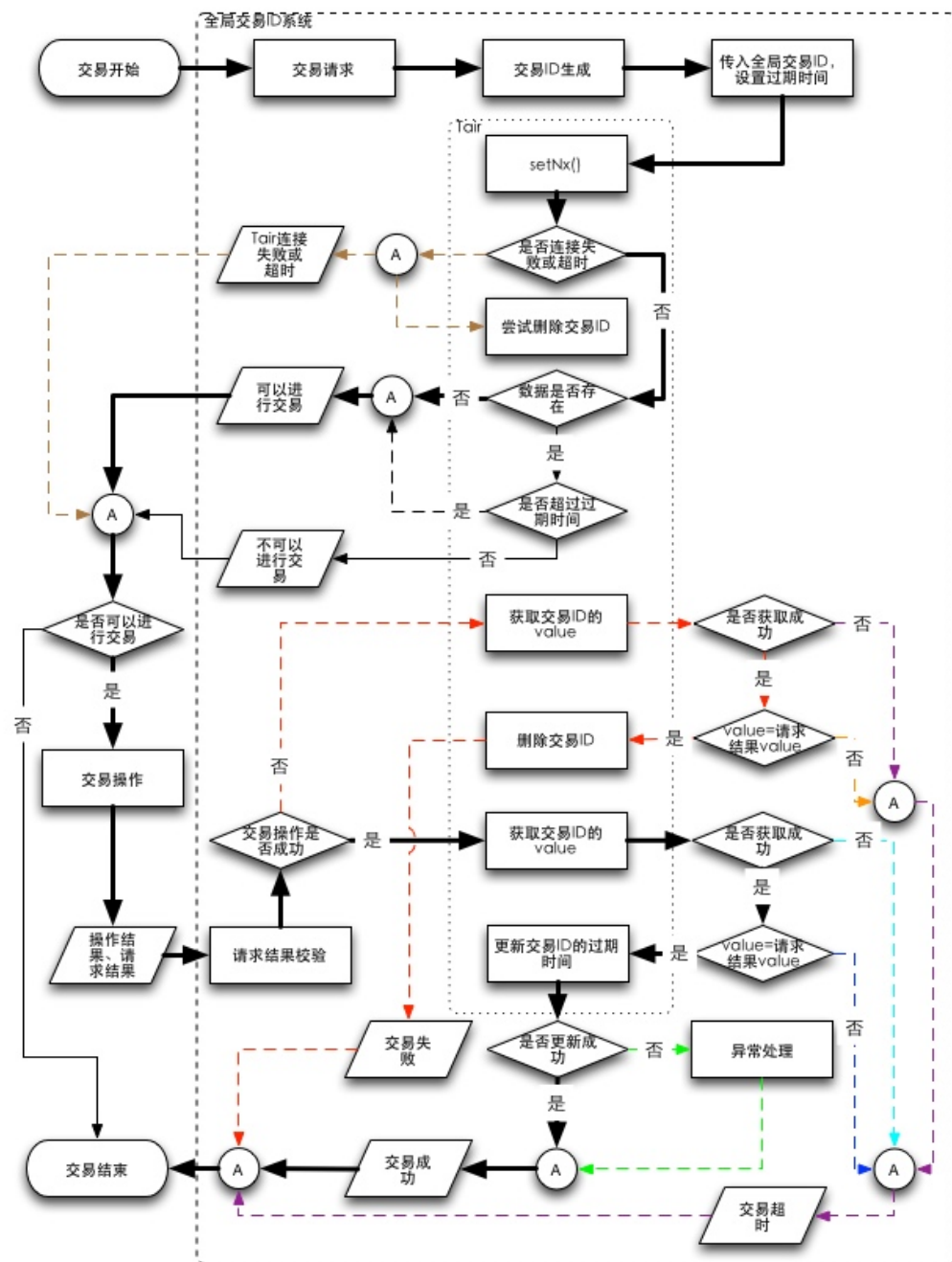
# GTIS-实现难点

---

从可用性的角度出发，我们需要考虑如下几点：

- 业务操作失败，需要重复执行的，引入业务失败删除机制
- 业务操作超时或者其他异常，需要引入超时机制
- 在进行失败删除机制时，需要保证不删除另外一个相同操作产生的ID





# GTIS-接口说明-前置接口

---

PreResult preGlobalTransIDRequest(Object transContents, int expireTime, int faultPolicy, Long timeOut, int retry)

操作前调用；返回是否允许操作继续进行的结果；已经进行过的相同操作将不被允许继续进行  
参数：

- transContents - 业务方输入操作唯一性的内容特性，可以是唯一性的String类型的ID，也可以是map、POJO
- expireTime - transContents的过期时间，业务方预估的大于操作时间的值，单位为秒，可选，默认3600s
- faultPolicy - 故障处理策略，连接失败或超时等故障，业务处理方式两种，可选，默认为DirectReturnPolicy：
- timeOut - 超时时间，业务方期望的调用Tair的方法所用的时间，单位为毫秒，可选，默认200ms
- retry - 重试次数



# GTIS-接口说明-后置接口

---

FinalResult finalGlobalTransIDProcess(PreResult result, int transResult, int expireTime, Long timeOut, int retry)

操作后调用；在业务方操作之后进行处理，返回最终操作结果，建议将该方法放入finally体中。

参数：

- result - preGlobalTransIDRequest返回的PreResult，业务方直接将返回的PreResult传入该方法即可。
- transResult - 业务方输入的操作是否成功的结果，即是否希望有第二次相同的操作可以执行；
- expireTime - 存活时间，该次成功操作的最终保持时间，单位为秒，可选，默认为604800s（7days），建议根据实际业务情况选择保持的时间。0为永不过期；小于当前时间为相对时间；大于当前时间为绝对时间戳。其他则为默认时间。
- timeOut - 超时时间，业务方期望的调用Tair的方法所用的时间，单位为毫秒，可选，默认200ms
- retry - 重试次数





# GTIS-接口说明-使用方法

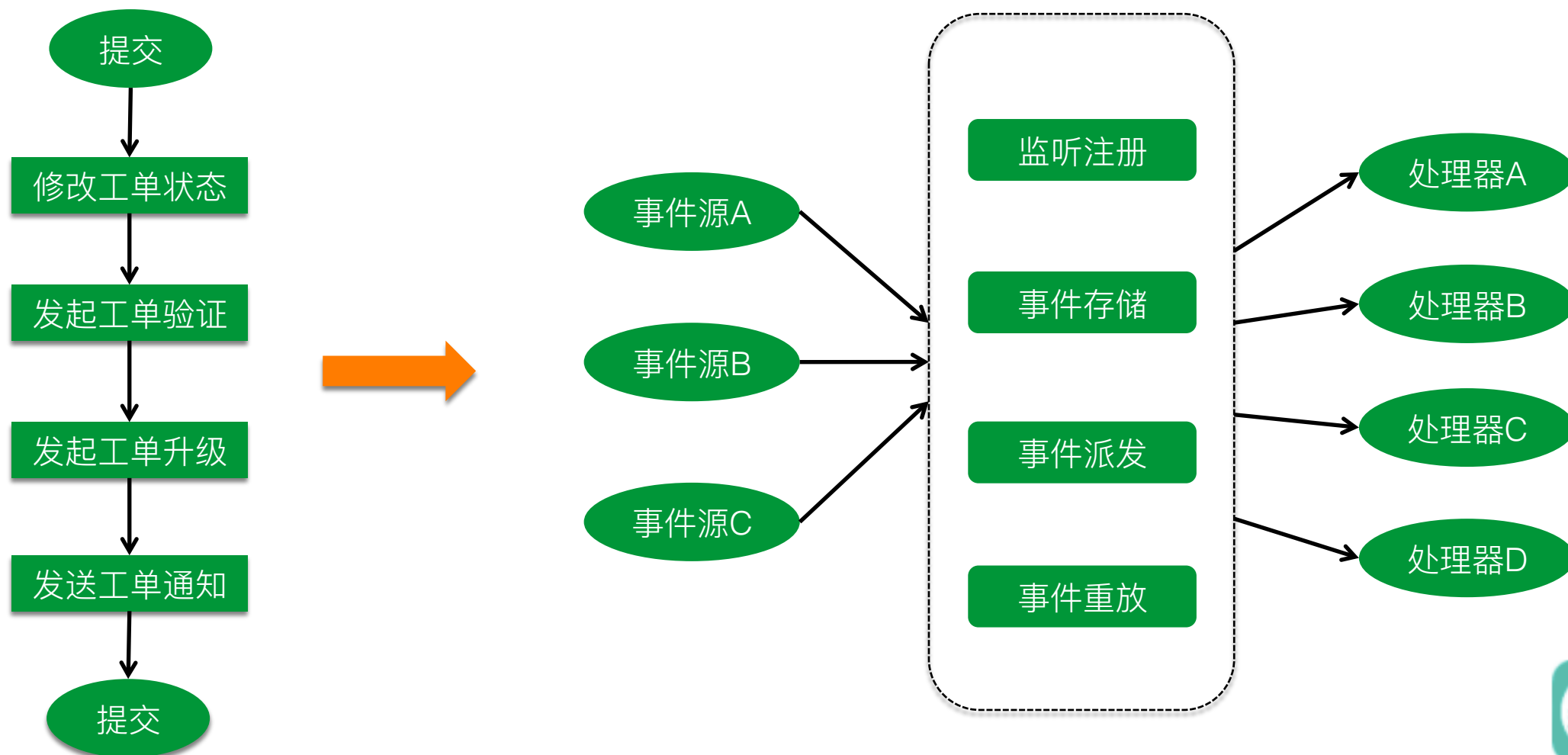
.....

```
Order order = new Order(s: "1234", hotelA: "hotelA", i: 250);

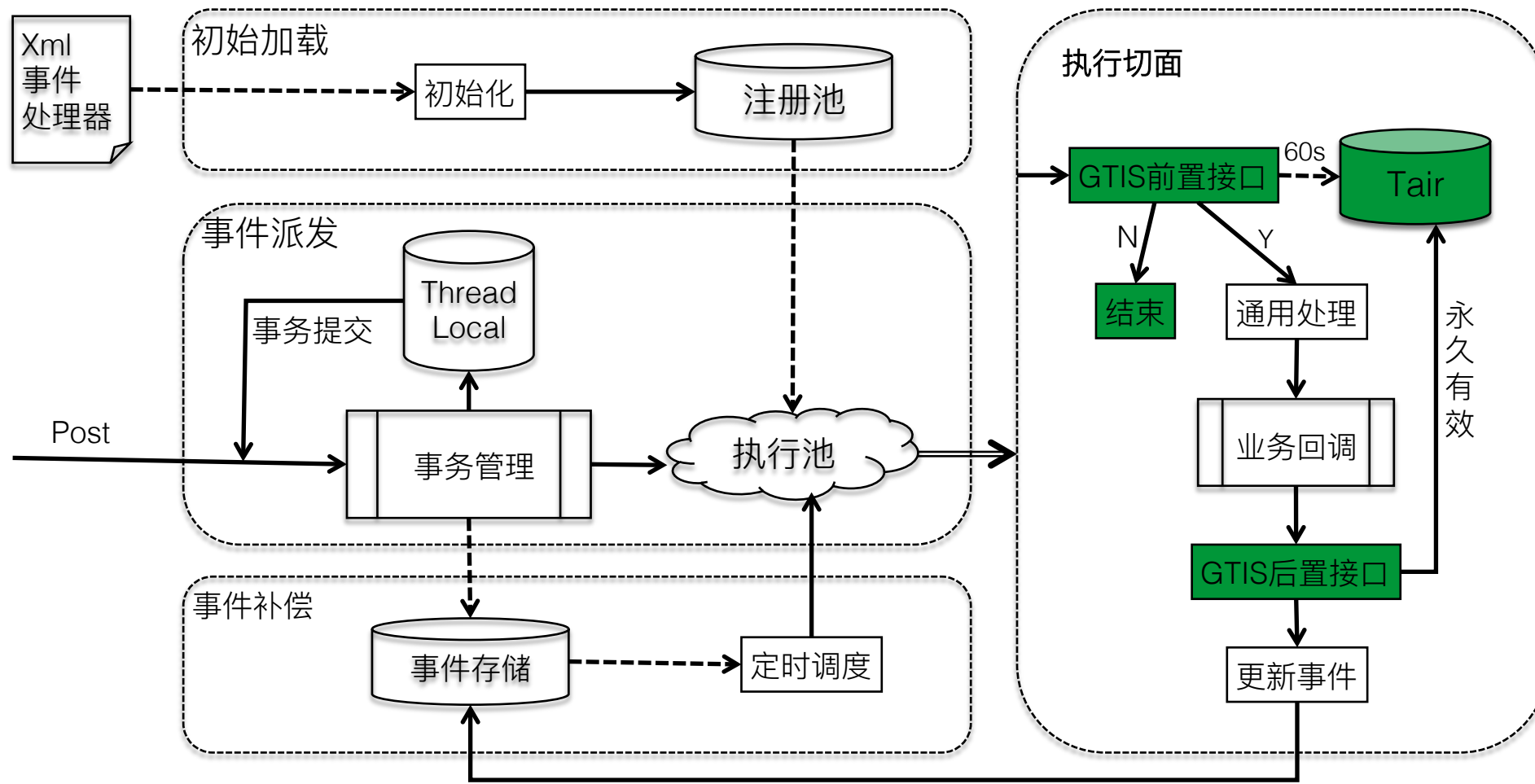
PreResult preResult = gtisService.preGlobalTransIDRequest(order, expireTime: 600); // 10min
int transResult = TransResultEnum.FAILED.getCode();
try {
    if (preResult.getErrCode() == PreResultEnum.TRANS_PERMITTED.getCode()) { // 表明可以进行操作
        // 进行业务操作
        transResult = doSomething();
    } else if (preResult.getErrCode() == PreResultEnum.TRANS_DENIED.getCode()) { // 表示是重复操作
        doNothing();
    }
} finally {
    gtisService.finalGlobalTransIDProcess(preResult, transResult, expireTime: 604800); // 7days
}
```



# GTIS-使用案例-EDA



# GTIS-使用案例-EDA



使用Gtis  
保证事件  
补偿的幂  
等性

# GTIS-现状

---

至今

- GTIS已持续迭代9个版本
- 稳定运行在了美团点评酒旅事业群的8个项目中：
  - 酒旅平台结算-防止重复下载对账单
  - 酒旅平台结算-防止重复挂账
  - 酒旅销售工单-事件回放幂等
  - ...



Let's Recap



# CAP in Distributed System



Consistency: Strong or Eventual



# Mutex





# Distributed Lock: Cerberus



# Idempotent



Gtis



The Price of Evolution is--  
Out Of Control

*-Kevin.Kelly*



给时光以生命，给岁月以文明

欢迎简历来投  
liuyunpeng@meituan.com

